Abstract Interpretation

(Cousot, Cousot 1977)

also known as

Data-Flow Analysis

(Kildall 1973)

Example: Constant propagation

Here is why it is useful

a → ⊥ b → ⊥
step → ⊥
i → ⊥ = ∅

```
int a, b, step, i;
boolean c;
a = 0;            → a → {0}
b = a + 10;       → b → {10}
step = -1;        → step → {-1}
if (step > 0) {
  i = a;          → i → {0}
} else {
  i = b;          → i → {10}
}
c = true;                          i → ⊤
while (c) {
  print(i);
  i = i + step;    // can emit decrement
  if (step > 0) {
    c = (i < b);
  } else {
    c = (i > a); // can emit better instruction here
  } // insert here (a = a + step), redo analysis
}
```

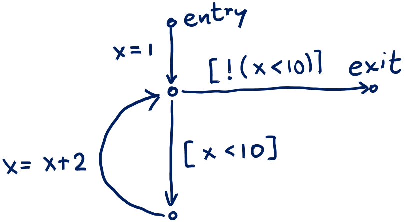
Goal of Data-Flow Analysis

Automatically compute information about the program

- Use it to report errors to user (like type errors)
- Use it to optimize the program

Works on control-flow graphs:
(like flow-charts)

```
x = 1
while (x < 10) {
  x = x + 2
}
```

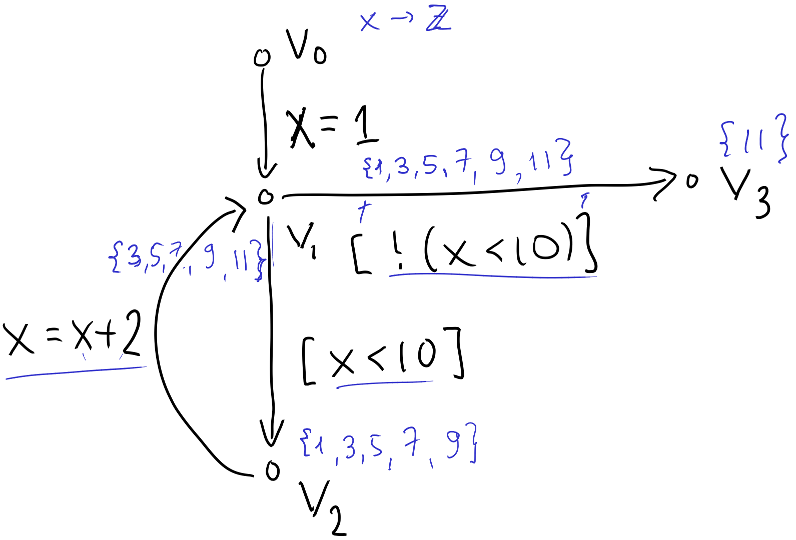


Interpretation and Abstract Interpratation

- Control-Flow graph is similar to AST
- We can
  - interpret control flow graph
  - generate machine code from it (e.g. LLVM, gcc)
  - abstractly interpret it: do not push values, but **approximately compute supersets of possible values** (e.g. intervals, types, etc.)

Compute Range of x at Each Point

$\mathbb{Z}$



What we see in the sequel

1. How to compile abstract syntax trees into control-flow graphs
2. Lattices, as structures that describe abstractly sets of program states (facts)
3. Transfer functions that describe how to update facts
4. Basic idea of fixed-point iteration

Generating Control-Flow Graphs

- Start with graph that has one entry and one exit node and label is entire program
- Recursively decompose the program to have more edges with simpler labels
- When labels cannot be decomposed further, we are done

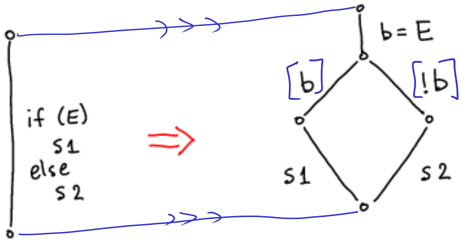
Flattening Expressions
for simplicity and ordering of side effects

E_1, E_2 - complex expressions

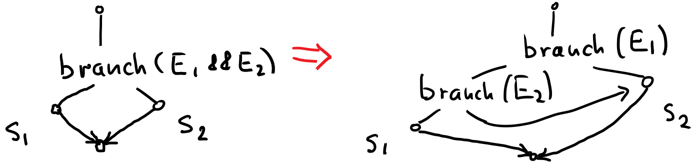
t_1, t_2 - fresh variables

$$x = E_1 * E_2 \quad \Longrightarrow \quad \begin{array}{l} t_1 = E_1 \\ t_2 = E_2 \\ x = t_1 * t_2 \end{array}$$

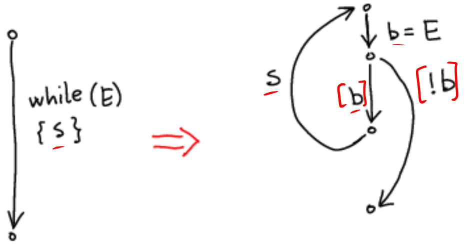
If-Then-Else



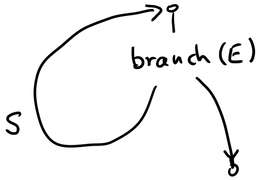
Translation using "branch" instruction works nicely for control flow graphs: two destinations



While



Better translation uses the "branch" instruction



Example 1: Convert to CFG

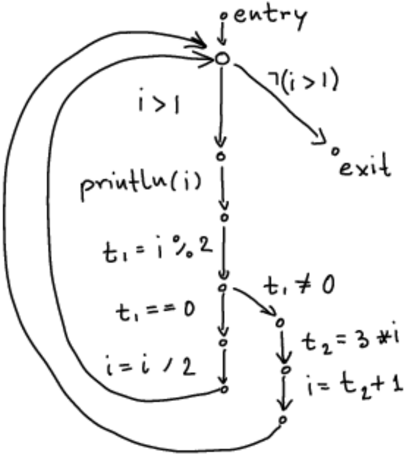
```
while (i < 10) {
  println(j);
  i = i + 1;
  j = j +2*i + 1;
}
```

Example 2: Convert to CFG

```
int i = n;
while (i > 1) {
  println(i);
  if (i % 2 == 0) {
    i = i / 2;
  } else {
    i = 3*i + 1;
  }
}
```

Example 2 Result

```
int i = n;
while (i > 1) {
  println(i);
  if (i % 2 == 0) {
    i = i / 2;
  } else {
    i = 3*i + 1;
  }
}
```



Translation Functions

$[\ s_1\ ;\ s_2\ ]\ v_{source}\ v_{target} =$

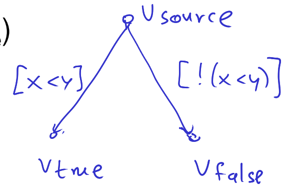
$\quad [\ s_1\ ]\ v_{source}\ v_{fresh}$

$\quad [\ s_2\ ]\ v_{fresh}\ v_{target}$

$[$ **branch**$(x<y)\ ]\ v_{source}\ v_{true}\ v_{false} =$

insert$(v_{source},[x<y],v_{true})$;

insert$(v_{source},[!(x<y)],v_{false})$



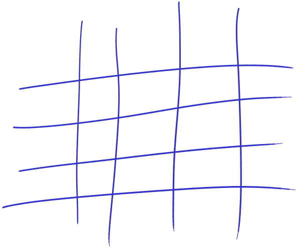
insert $(v_s,stmt,v_t)=$

$cfg = cfg + (v_s,stmt, v_t)$

$[\ x=y+z\ ]\ v_s\ v_t =$ **insert**$(v_s,x=y+z, v_t)$

when y,z are constants or variables

Abstract Interpretation: Analysis Domain (D) Lattices



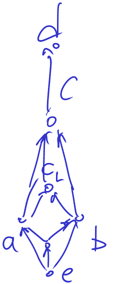
Lattice

$\subseteq$

Partial order: binary relation $\leq$ (subset of some D^2) which is

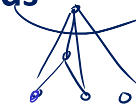
- reflexive: $x \leq x$
- anti-symmetric: $x \leq y \wedge y \leq x \rightarrow x = y$
- transitive: $x \leq y \wedge y \leq z \rightarrow x \leq z$

closure of directed acyclic
acyclic



Lattice is a partial order in which every **two-element** set has **least among its upper bounds** and **greatest among its lower bounds**

$a \leq c$
$b \leq c$



- Lemma: if $(D, \leq)$ is lattice and D is finite, then lub and glb exist for every finite set

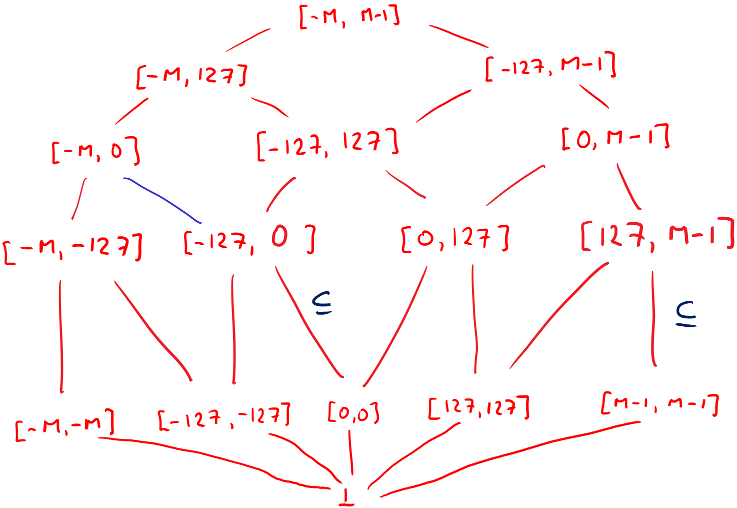
$\sqcap \quad \sqcup \qquad \sqcup \{a, b, c\}$

Graphs and Partial Orders

- If the domain is finite, then partial order can be represented by directed graphs
  - if $x \leq y$ then draw edge from x to y
- For partial order, no need to draw $x \leq z$ if $x \leq y$ and $y \leq z$. So we only draw non-transitive edges
- Also, because always $x \leq x$, we do not draw those self loops
- Note that the resulting graph is acyclic: if we had a cycle, the elements must to be equal

Domain of Intervals [a,b] where $a,b \in \{-M,-127,0,127,M-1\}$

$M = 2^{31}$



Defining Abstract Interpretation

Abstract Domain D describing which information to compute – this is often a lattice

- inferred types for each variable: x:T1, y:T2
- interval for each variable   x:[a,b], y:[a',b']

Transfer Functions, [[**st**]] for each statement **st**, how this statement affects the facts $D \to D$

- Example:

$[\![x = x+2]\!](x:[a,b], \dots)$

$= (x:[a+2, b+2], \dots)$

$x:[a,b] \quad y:[c,d]$

$x = x+2$

$x:[a+2, b+2], y:[c,d]$

For now, we consider arbitrary integer bounds for intervals

[1, 9]
1,3,5,7,9

- Thus, we work with BigInt-s
- Often we must analyze machine integers
  - need to correctly represent (and/or warn about) overflows and underflows
  - fundamentally same approach as for unbounded integers
- For efficiency, many analysis do not consider arbitrary intervals, but only a subset of them W
- We consider as the domain
  - empty set (denoted ⊥ , pronounced "bottom")
  - all intervals [a,b] where a,b are integers and $a \leq b$, or where we allow $a = -\infty$ and/or $b = \infty$
  - set of all integers $[-\infty, \infty]$ is denoted T , pronounced "top"

Find Transfer Function: Plus

Suppose we have only two integer variables: x,y

$x: [a,b] \quad y: [c,d]$

$x = x+y$

$x: [a',b'] \quad y: [c',d']$

If $a \le x \le b \qquad c \le y \le d$

and we execute **x= x+y**

then $x' = x+y$ $\quad \to x = x' - y$

$y' = y$

$a \le x \le b$

$a \le x' - y \le b$

so

$a+c \le a+y \le x' \le b+y \le b+d$

$a+c \le x' \le b+d$

$c \le y' \le d$

So we can let

$a' = a+c \qquad b' = b+d$

$c' = c \qquad d' = d$

Find Transfer Function: Minus

Suppose we have only two integer variables: x,y

$x : [a, b] \quad y : [c, d]$

$y = x - y$

$x : [a', b'] \quad y : [c', d']$

If

and we execute **y= x-y**

then

$y' = x - y$

$y = x - y'$

$c \le y \le d$

$c \le x - y' \le d$

$c \le x - y'$

$y' \le x - c \le b - c$

So we can let

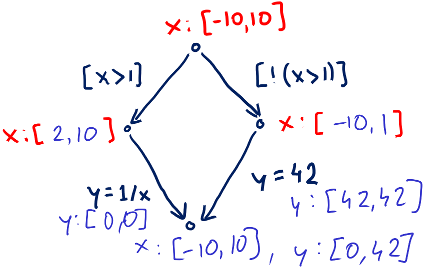
a’= a b’ = b

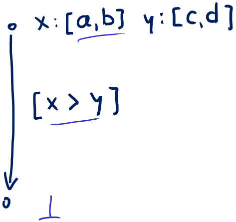
c’= a - d d’ = b - c

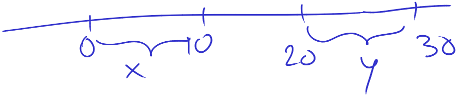
Transfer Functions for **Tests**

Tests e.g. [x>1] come from translating if,while into CFG

```
x : [-10,10]
if (x > 1) {
  x : [2, 10]
  y = 1 / x
} else {
  x : [-10, 1]
  y = 42
}
```







Joining Data-Flow Facts

x: [-10,10]   y: [-1000,1000]

if (x > 0) {

x: [1, 10]   y: [-1000, 1000]

y = x + 100

x: [1, 10]   y: [101, 110]

} else {   [x ≤ 0]

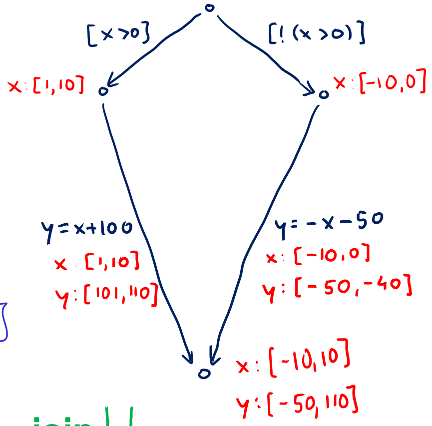
x: [-10, 0]   y: [-1000, 1000]

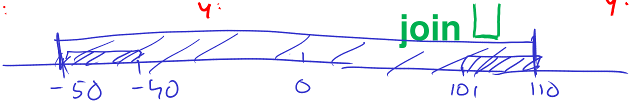
y = -x – 50

x: [-10, 10]   y: [-50, -40]

}

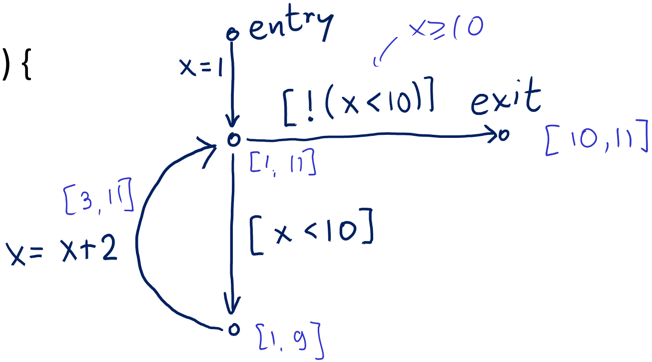
x: y:





Handling Loops: Iterate Until Stabilizes

```
x = 1
[1,1]
while (x < 10) {
  [1,1]
  x = x + 2
  [3,3]
}
```

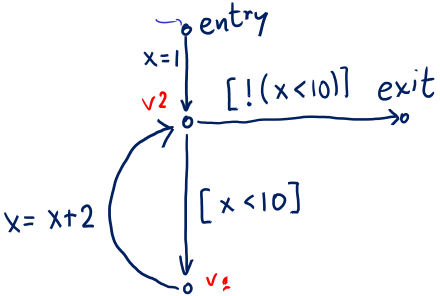


Analysis Algorithm

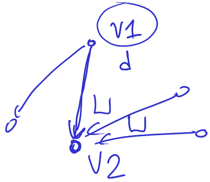
```
var facts : Map[Node,Domain] = Map.withDefault(empty)
facts(entry) = initialValues
while (there was change)
   pick edge (v1,statmt,v2) from CFG
            such that facts(v1) has changed
   facts(v2)=facts(v2) join transferFun(statmt, facts(v1))
}
```

$\sqcup$

Order does not matter for the end result, as long as we do not permanently neglect any edge whose source was changed.



```
var facts : Map[Node,Domain] = Map.withDefault(empty)
var worklist : Queue[Node] = empty
def assign(v1:Node,d:Domain) = if (facts(v1)!=d) {
  facts(v1)=d
  for (stmt,v2) <- outEdges(v1) { worklist.add(v2) }
}
assign(entry, initialValues)
while (!worklist.isEmpty) {
  var v2 = worklist.getAndRemoveFirst
  var update = facts(v2)
  for (v1,stmt) <- inEdges(v2)
    { update = update join transferFun(facts(v1),stmt) }
  assign(v2, update)
}
```



Work List Version

Exercise: Run range analysis, prove that **error** is unreachable

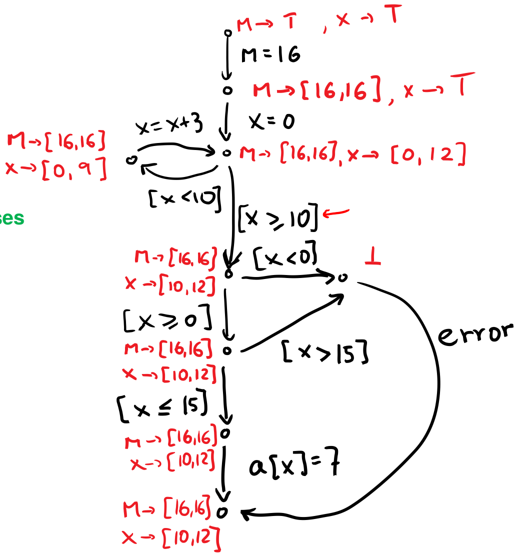
```
int M = 16;
int[M] a;
x := 0;
while (x < 10) {
  x := x + 3;
}
if (x >= 0) {
  if (x <= 15)
    a[x]=7;
  else
     error;
} else {
   error;
}
```

checks array accesses

Range analysis results

```
int M = 16;
int[M] a;
x := 0;
while (x < 10) {
  x := x + 3;
}
if (x >= 0) {
  if (x <= 15)
    a[x]=7;
  else
    error;
} else {
  error;
}
```

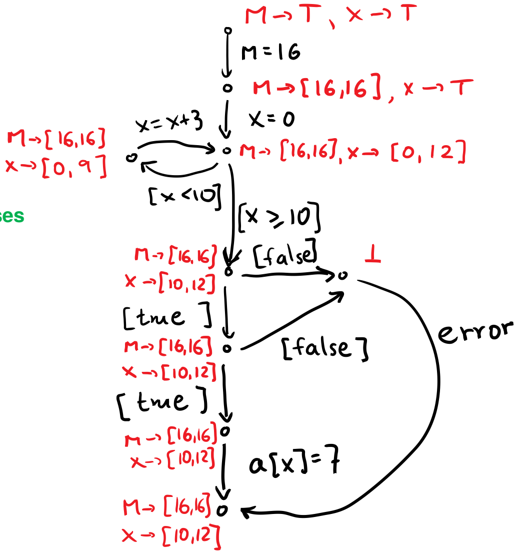
checks array accesses



Simplified Conditions

```
int M = 16;
int[M] a;
x := 0;
while (x < 10) {
  x := x + 3;
}
if (x >= 0) {
  if (x <= 15)
    a[x]=7;
  else
    error;
} else {
  error;
}
```

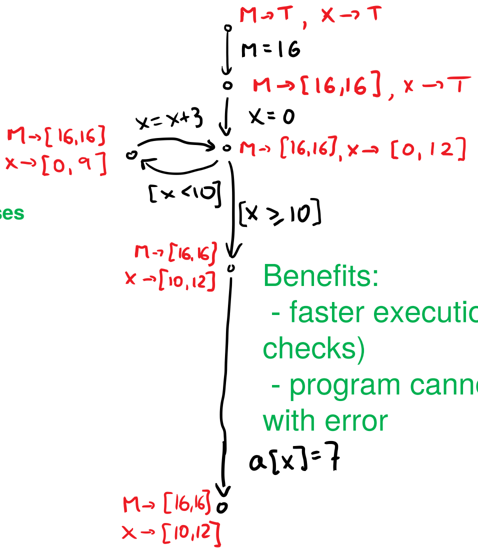
checks array accesses



Remove Trivial Edges, Unreachable Nodes

```
int M = 16;
int[M] a;
x := 0;
while (x < 10) {
  x := x + 3;
}
if (x >= 0) {
  if (x <= 15)
    a[x]=7;
  else
     error;
} else {
   error;
}
```

checks array accesses



Benefits:
- faster execution (no checks)
- program cannot crash with error

Constant Propagation Domain

Domain values D are:

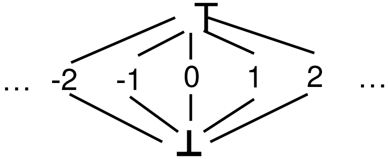
– intervals [a,a], denoted simply ‘a’

– empty set, denoted $\perp$ and set of all integers T

Formally, if **Z** denotes integers, then

$$D = \{\perp, T\} \cup \{\, a \mid a \in \mathbf{Z} \}$$

D is an infinite set



Constant Propagation Transfer Functions

v_1

x = y + z

v_2

$S_x + S_y$
$= \{x+y \mid x \in S_x, y \in S_y\}$

table for +:

$z \backslash y$	$\perp$	c_y	$\top$
$\perp$	$\perp$	$\perp$	$\perp$
c_z	$\perp$	c_y+c_z	$\top$
$\top$	$\perp$	$\top$	$\top$

For **each variable** (x,y,z) and
each CFG node (program point)
we store: $\perp$, a constant, or $\top$

```
abstract class Element
case class Top extends Element
case class Bot extends Element
case class Const(v:Int) extends Element
var facts : Map[Nodes,Map[VarNames,Element]]
```

what executes during analysis of x=y+z:

```
oldY = facts(v1)("y")
oldZ = facts(v1)("z")
newX = tableForPlus(oldY, oldZ)
facts(v2) = facts(v2) join facts(v1).updated("x", newX)
```

```
def tableForPlus(y:Element, z:Element)
= (x,y) match {
  case (Const(cy),Const(cz)) =>
                          Const(cy+cz)
  case (Bot,_) => Bot
  case (_,Bot) => Bot
  case (Top,Const(cz)) => Top
  case (Const(cy),Top) => Top
}
```